Stateless: an Optic Algebra Library

Jesús López-González
j.lopezgo@alumnos.urjc.es

Juan Manuel Serrano
juanmanuel.serrano@urjc.es

Javier Fuentes
javier.fuentes@hablapps.com

June 9, 2017

Abstract

Abstract goes here!

1 Introduction

Software industry as we know it lacks well-founded frameworks to program reactive systems in a modular way. This kind of systems hide an inner *state* to their environment, showing therefore a black-box behaviour. Take a university information system as a guiding example. Universities are structured in departments, and each has a budget. Even this simple definition does contain three kinds of entities: universities, departments and budget. Ideally, programming the evolving state of the whole system should be achieved by programming the state of each entity or subsystem, and finally composing them all.

Provide solid references!

In addition, composition should contemplate heterogeneity: departments could be stored in a local database while budget could be deployed as external microservices. Ultimately, any configuration should be supported, but it is essential for the system description to remain pure and decoupled from low-level details, irrelevant to the domain. Needless to say that it should never hamper optimal performance. Programming a declarative and modular system in this way is still a great challenge, but functional programming seems like a good starting point [Hughes, 1989].

There are plenty of technologies to develop reactive systems, but they are postulated as solutions to very specific problems, and consequently, difficult to be reused. Turning to the university example, we could have decided to program department evolution on a relational database, like *MySQL*. However, if large volumes of data arose in time, migrating to a distributed database such as *Redis* might become necessary. Given that situation, we would find it extremely difficult to adapt existing encodings to the new setting.

Fortunately, industry has detected this issue and new abstract tools have emerged to fill the gap. In this sense, we could have avoided the aforementioned problem by programming department evolution with *Slick*, a Functional-Relational Mapper that provides an *embedded domain specific language* [Hudak, 1996] which can be later interpreted into different databases. Similarly, we could find other situations where a common language could unify different state evolutions. Take *optics* as an example. They could be seen as a language to achieve complex transformations over immutable in-memory data. In this scenario, *Traversal* [O'Connor, 2011] could be used as the glue to keep university and department related.

We claim that many of these languages to access state could be further factorized, by providing a new language to unify them, while retaining reasonable levels of performance for a large class of applications. Moreover, we claim that optic concepts are ideal to abstract technology-specific details away. However, prior to that, they have to be taken to a more general setting: *optic algebras*.

An optic algebra represents the essence of the state transformation associated to that optic. Generally speaking, optics were conceived to tackle a very specific problem, achieving complex transformations over immutable in-memory data structures. For instance, a *lens* [Foster et al., 2005] knows how to transform certain data structure —the focus— which is contextualized in a larger one —the whole. Regarding our university, this abstraction could perfectly describe the existing relationship between a department and its associated budget. Having said so, what do we mean by the essence of a lens?

The essence of a lens requires removing all those aspects which are specific to in-memory transformations. If we take into account the classic definition of a lens in Scala [Morris, 2012]:

I'd like to avoid showing snippets in the intro!

$$(s \to a,\ a \to s \to s)$$

one could easily identify several accesses to the in-memory state S. If we abstract away from those aspects, ie. if we conceal the state, we end up with the optic algebra associated to lens:

$$(s \to f\ a,\ a \to f\ ())$$

This algebra —which bears a strong resemblance to *MonadState* [Gibbons, 2012]— determines the existence of a focus that could be either read or updated, but it keeps itself unaware of the details underneath F. Similar derivations could be carried out over other optics, resulting in new optic algebras.

Composing optic algebras —as they were introduced in the previous paragraph— is not straightforward at all. In fact, the situation is very similar to the one that emerges when trying to compose optics in their classic representation. In this sense, Van Laarhoven and recently Profunctor optics [Pickering et al., 2017] have been postulated as alternative representations to improve composition. Taking them as inspiration, we created a new representation for optic algebras. Since it heavily relies on natural transformations [Pierce, 1991], we refer to it as *natural* representation. This representation not only improves the composition up, but also introduces the notion of heterogeneity, as we will see in further sections.

Optic algebra instances for the most widespread deployments should be provided. Given this scenario, by defining our system in terms of optic algebras, we should be able to get our logic implemented once and for all. Later, we could find it interpreted into a method call, a database access, a microservice invocation, or even as a combination of many of them.

We claim that optic algebras in their natural representation lead to modular software designs. For instance, a traversal algebra could establish the connection between the university and each department. On the other hand, a lens algebra could describe the relationship between departments and budget. Then, we could compose both algebras to conform the whole system. Additionally, optic algebras will enable us to do it in an heterogeneous way, with many free instances ready to go, offering a reasonable performance in the majority of cases.

2 Background

2.1 Monads

2.2 Optics

3 Deriving Optic Algebras

Previously, in section 2.2, optics were shown as crucial abstractions to access certain state, particularly, they provide access to in-memory data structures. Furthermore, we discussed that this effect could be perfectly modeled with the *State Monad*. Thereby, optics enable programmers to describe a whole range of complex transformations as State programs. Having said so, it would be great to take this potential to other *stateful* scenarios.

Optic algebras distill optics to extract their algebraic essence. More specifically, they abstract all the in-memory specific details away, to make the optic potentially exploitable in new contexts, such as relational databases, microservices or event sourcing, among others. The remainder of this section will turn the most prominent optics into their corresponding algebraic representation, beginning with lens.

3.1 Lens Algebra

How could one extract the essence of a lens? What does it mean to abstract the in-memory details away? Well, we know that lenses can be represented as a pair of functions: *get* and *set*. Those functions are aware of the existence of an outer state, and they use it to extract the focus from, and to set a new value for the focus to. Our main objective here is to abstract away any reference to that outer state. We do so with the following steps:

Proof. Let s, a be types representing the whole and the focus of certain lens, respectively. So,

This proof is not a proof at all, lier!

$$
\begin{aligned}
&(get, set) \\
&\quad = (s \to a,\ a \to s \to s) && \text{(expand definitions)} \\
&\quad = (() \to s \to a,\ a \to s \to s) && \text{(homogenize input)} \\
&\quad = (() \to s \to a,\ a \to s \to (s, ())) && \text{(homogenize output)} \\
&\quad = (() \to s \to (s, a),\ a \to s \to (s, ())) && \text{(homogenize resulting state *)} \\
&\quad = (() \to State\ s\ a,\ a \to State\ s\ ()) && \text{(State isomorphism)} \\
&\quad \therefore (() \to f\ a,\ a \to f\ ()) && \text{(Abstract away)}
\end{aligned}
$$

□

The first step after expanding definitions consists of making the input in both functions homogeneous. Since *set* receives an A as first parameter, something has to be added as a first argument in *get*. Thereby, by applying simple algebraic rules, we can safely prepend a redundant *Unit* typed parameter.

Secondly, we homogenize output. In this sense, *get* is returning an A as output, while *set* is only returning the updated value for S. If we want the latter to return its own output, we have to provide a redundant *Unit* accompanying the resulting state S.

Thirdly, once we have homogenized the output value, it should be desirable to homogenize the resulting state as well. To do so, we create a contrived S as companion of the output A, which will contain the second parameter as is. This step is tricky, since it is not a valid equivalence per se. Therefore, in order to formalize it, we have to impose that the resulting state has to be exactly the same as the incoming one, as a new law for our derived *get*.

Make this clear somehow in the corresponding step

Finally, it is not hard to see that the right hand side of the resulting functions corresponds with the state monad, being S the underlying state. Given this isomorphism, we can say we have our functions homogenized, both getting certain input and returning a stateful program. If you remember our initial objective, we wanted to abstract all those references to the outer state S away. Now, it is easy to do so, we just have to turn *State[S, ?]* into a generic type constructor *F[?]*. Hopefully, the resulting pair should be familiar to the reader.

We started our derivation trying to remove low-level details from our lens and we ended up with something really similar to *MonadState*. As you can see, we are lacking the monadic combinators to fully complete the connection. At first glance, this seems like a big gap, but you will be pleased to hear that we are not that far, we just need to bring lens laws into the picture.

Lens laws have an important role to connect the resulting algebra with *MonadState*. You must recall that a lens is not only a pair of functions, it is closely tied to the lens laws: *PutPut*, *GetPut* and *PutGet*. Therefore, they have to be taken into account in the derivation process, because they also mutate along with the pair of functions. Those adaptations are a walk in the park, but there is a particular point when the law adaptation no longer makes sense.

What does the *GetPut* law mean right after the last step? In fact, there is no way to establish a connection between the pair of functions at this level of f abstraction, unless we provide such mechanism with monadic combinators. Given this situation, it is reasonable to extend our algebra with *Monad*, because laws have to be adapted into this abstract setting. Luckily, we do not have to do such work, because this step has already be done. In fact, by adapting lens laws to the new monadic setting, we are just reinventing *MonadState* laws.

The lens and *MonadState* laws correspondence is straightforward, however *MonadState* has an additional intruder which is not contemplated in the lens laws: *GetGet*. So, we could think that *MonadState* is imposing additional restrictions to the ones that we adapted from lens laws. Nevertheless, we should recall that we added an additional law to our representation when we homogenized the resulting state. If we adapt it to the abstract setting, we end up with somethink similar to this:

$$get >> return\ () = return\ () \qquad \text{(XGet)}$$

It tells us that instructing a *get* and ignoring its result does not produce an overall effect. It would be great if *GetGet* could be derived from *XGet*. We show that this is possible:

Proof. GetGet can be derived from XGet

Any leap of faith in this process?

Monad laws steps are quite tedious

GetGet

$$\begin{aligned}
&= get >>= (a1 \to get >>= (a2 \to return\ (a1, a2))) = get >>= (a \to return\ (a, a)) && \text{(GetGet)}\\
&= get >>= (a1 \to get >>= (a2 \to return\ (a1, a2))) >> return\ () = \\
&\quad get >>= (a \to return\ (a, a)) >> return\ () && \text{(Monad laws)}\\
&= get >>= (a1 \to get >>= (a2 \to return\ ())) = get >>= (a \to return\ ()) && \text{(Monad laws)}\\
&= get >> get >> return\ () = get >> return\ () && \text{(Syntax)}\\
&= get >> return\ () = get >> return\ () && \text{(XGet)}
\end{aligned}$$

By establishing this connection we can say that the adaptation of lens laws, including the one we generated in the abstraction process, corresponds directly with *MonadState* laws. Thereby, we claim that the algebraic essence of a lens is exactly *MonadState*. In fact, this typeclass abstract away all those in-memory specific details while preserving the lens laws.

References

[Foster et al., 2005] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2005). Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246.

[Gibbons, 2012] Gibbons, J. (2012). Unifying theories of programming with monads. In *International Symposium on Unifying Theories of Programming*, pages 23–67. Springer.

[Hudak, 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196.

[Hughes, 1989] Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

[Morris, 2012] Morris, T. (2012). Asymmetric lenses in scala.

[O'Connor, 2011] O'Connor, R. (2011). Functor is to lens as applicative is to biplate: Introducing multiplate. *arXiv preprint arXiv:1103.2841*.

[Pickering et al., 2017] Pickering, M., Gibbons, J., and Wu, N. (2017). Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857*.

[Pierce, 1991] Pierce, B. C. (1991). *Basic category theory for computer scientists*. MIT press.